Partial Evaluation of Concurrent Programs

Matthieu Martel[1†] and Marc Gengler[2]

[1] CEA - Saclay, LIST-DTSI-SLA, F91191 Gif-sur-Yvette Cedex
mmartel@cea.fr
[2] Laboratoire d'Informatique de Marseille, 163, avenue de Luminy,
Case 901 F, 13288 Marseille Cedex 9, France
Marc.Gengler@esil.univ-mrs.fr

Abstract. In this article, we introduce a partial evaluator for a concurrent functional language with synchronous communications over channels, dynamic process and channel creation, and the ability to communicate channel names. Partial evaluation enables to execute at compile-time the communications of a program for which the emitter, the receptor and the message are statically known. The partial evaluator and the static analyses used to guide it were implemented and we show how to specialize concurrent programs for particular execution contexts, corresponding to assumptions on the network or on the messages.
Keywords: Partial Evaluation, Concurrent Languages, Binding-time Analysis, Control Flow Analysis.

1 Introduction

Partial evaluation is a technical tool that enables to execute a program for which only one part of the data is known [9]. Only the *static* instructions, depending on the known data, are executed, while the *dynamic* intructions, depending on unknown data, are frozen. We obtain a new program, made of the pieces of code which could not be executed, and of the result of the achieved computations. To determine the parts of a program which can be executed because they solely depend on known data, partial evaluators usually use the results of a static analysis called *binding-time analysis* (BTA) [3, 9].

Even though partial evaluation techniques have been widely studied for sequential languages, partial evaluation of concurrent programs has received little attention. However, concurrency introduces new constructs for which specific methods must be developed. Hosoya *et al.* have proposed an *on-line* partial evaluator for a concurrent language close to ours [8]. On-line partial evaluators do not use the results of a BTA in order to improve their accuracy. Marinescu and Goldberg have proposed a partial evaluator for a CSP-like language with static channels [10]. Gengler and Martel have proposed a partial evaluator for the π-calculus as well as sufficient conditions on the annotations to ensure the correctness of the program output by partial evaluator wrt. the original one [5,

[†] Work done while the author was at Laboratoire d'Informatique de Marseille.

11]. Solberg *et al.* and Bodei *et al.* have proposed a control flow analysis (CFA) useful to improve the precision of the BTA [2, 20]. This analysis was improved in [12,13].

In this article, we are interested in partial evaluation of a concurrent functional language. The operational semantics is based on the one of the language λ_{cv} proposed by Reppy for Concurrent ML [1, 17, 18]. This language offers synchronous communications over channels, dynamic process creation and the ability to communicate channel names and functions. In this context, partial evaluation enables to statically execute the communications of a program for which the emitter, the receptor and the content of the message are known [5, 8, 10]. We obtain a residual program with fewer communications than the original one.

We introduce a partial evaluator, denoted `Pev` which uses the result of a BTA and of a control flow analysis (CFA) [16, 19] in two different ways. First, to determine the functions possibly called at a given application point, and, second, to know the possible synchronizations of the program, i.e. the possible matching pairs of emitter and receptors. The CFA of [13], the BTA and the partial evaluator were implemented and we describe some experiments. We show how to specialize, by partial evaluation, concurrent programs with respect to particular execution contexts: static knowledge on the topology of the network, assumptions on the behavior of the network, or assumptions on the transmitted data. In all cases, we show that `Pev` outputs residual programs with less communications than the original ones.

This article is organized as follows. Section 2 gives the principles of the binding time and control flow analyses used to annotate the programs provided to the partial evaluator. Section 3 introduces the partial evaluator `Pev`. In Section 4, we show how to specialize concurrent programs with `Pev`.

2 Program annotations

A binding-time analysis determines which instructions depend on the known data and can be executed by a partial evaluator. For concurrent languages with explicit communications, a BTA has to determine which communications are static, that is, occur on channels known at partial evaluation-time and transmit static data. Static communication are executed at partial evaluation-time while dynamic communications are left unchanged in the residual program output by `Pev`. In order to efficiently annotate a program, a BTA has to precisely know the *topology of the communications*, ie. the pairs of possibly matching emitters and receptors.

The BTA we use is defined in [11]. For the sequential part of the language, it is a usual BTA based on the one introduced by Bondorf and Jorgensen [3]. However, it uses the topological information provided by a control-flow analysis (CFA) [13, 16], specially designed to improve the accuracy of the binding-time annotations at communication points. This CFA is defined in [11, 13] and works as follows.

Obviously, the communication topology of a concurrent program depends on the way the communications are ordered on each process. Hence, for each process in the system, we build a finite automaton which indicates how the communication points possibly follow each other. A transition related to a sequential reduction step is labeled ε and we introduce a ℓ-transition to denote the occurrence of a communication point labeled ℓ. The result is an approximation of the communication scheme for each process.

Knowing how the communication points are ordered on each process, we have to determine (an approximation of) their interactions. We introduce a reduced product automaton RA which is polynomial in the size of the source program and which provides an approximation of the possible interactions between processes.

Intuitively, the only informations relevant to the analysis are the pairs of emitter and receptor which possibly interact together. For instance, we want to know the set of emission points able to communicate with a given reception point, independently of any ordering, i.e. of the location of this communication in a trace of the program execution. The standard product automaton contains all the possible sequences of communications, and is too precise for our purpose. RA is able to determine the points in a program which possibly are active once a particular communication occurs. Let ℓ_s and ℓ_r be two points corresponding to both components of a communication c. A state q_c denotes c in RA. We attach to this state the set of points $\mathcal{S}(q_c)$ possibly active after execution of c. A second communication c' between two new points ℓ'_s and ℓ'_r is allowed if $\ell'_s \in \mathcal{S}(q_c)$ and $\ell'_r \in \mathcal{S}(q_c)$. In this case, we add a c-transition from q_c to $q_{c'}$ and $\mathcal{S}(q_{c'})$ is updated. So, if two different communications c_1 and c_2 can follow c, then two c-transitions link q_c to q_{c_1} and q_{c_2}. Doing so, RA indicates that c_1 and c_2 are two possible communications without specifying their ordering. The size of the reduced product automaton we obtain is $O(n^4)$ where n is the length of the program to be analyzed.

We conclude this Section by emphasing the existing links between the BTA and the CFA. As indicated above, the precision of the BTA depends on the one of the topology provided by the CFA. Let us consider a program made of two processes realizing a static and next a dynamic communication on the same channel γ. The results produced by our BTA for this program are given in Equation (1), in which underlined operations are dynamic.

$$\texttt{let } p_1 = \texttt{fork} \begin{pmatrix} \texttt{let } s_1 = \texttt{send } \gamma\ S \texttt{ in} \\ \texttt{let } s_2 = \underline{\texttt{send}}\ \gamma\ D \texttt{ in} \\ \ldots \end{pmatrix} \texttt{ in } \begin{pmatrix} \texttt{let } r_1 = \texttt{receive } \gamma \texttt{ in} \\ \texttt{let } r_2 = \underline{\texttt{receive}}\ \gamma \texttt{ in} \\ \ldots \end{pmatrix} \quad (1)$$

We can see that only the second communication is annotated dynamic. This is due to the fact that the topological informations allow, in this case, to determine the exact pairs of emitters and receptors [13]. A less precise BTA, for instance based on the topological informations provided by the analysis of [20], would annotate both communications as being dynamic since γ is once used to communicate a dynamic value.

3 Partial evaluation

In this Section, we introduce the language used by the partial evaluator, denoted `Pev`, and we define the partial evaluator.

The language used by `Pev` is based on the language λ_{cv} introduced by Reppy [17]. It is an untyped subset of Concurrent ML [1,18] with dynamic process and channel creations and synchronous communications over channels. Just like regular values, channel names created by the instruction `channel()` and functions are ground values which may be communicated. The syntax is defined by

$$\begin{array}{l} e ::= v \mid x \mid \texttt{fun}\ x\ \texttt{=>}\ e_0 \mid \texttt{rec}\ f\ x\ \texttt{=>}\ e_0 \mid e_0\ @\ e_1 \mid \texttt{if}\ e_0\ e_1\ e_2 \\ \quad \mid \texttt{channel()} \mid \texttt{fork}\ e_0 \mid \texttt{send}\ e_0\ e_1 \mid \texttt{receive}\ e_0 \end{array} \tag{2}$$

This language contains conditionals and the operator `rec` for recursive functions. `channel()` denotes a function call which creates and returns a new channel name k, different from all the existing ones. `fork` e_0 creates a new process which computes e_0. `send` $e_0\ e_1$ is the emission of the value of e_1 on the channel resulting from the evaluation of e_0. e_0 and e_1 respectively are the *subject* and the *object* of the communication. `receive` e_0 is the reception of a value on the channel name described by e_0 (the subject of the reception). Values are in the domains of basic types, or channel names, or functions.

As stated earlier, we annotate the programs to indicate to `Pev` which expressions can be executed at partial evaluation-time. This is done by mean of a *two-level language* [6,15], extending the grammar of Equation (2) in the following way.

$$\begin{array}{l} e ::= c \mid x \mid \texttt{fun}\ x\ \texttt{=>}\ e_0 \mid \texttt{rec}\ f\ x\ \texttt{=>}\ e_0 \mid e_0\ @\ e_1 \mid \texttt{if}\ e_0\ e_1\ e_2 \\ \quad \mid \texttt{channel()} \mid \texttt{fork}\ e_0 \mid \texttt{send}\ e_0\ e_1 \mid \texttt{receive}\ e_0 \\ \quad \mid \underline{c} \mid \underline{\texttt{fun}}\ x\ \texttt{=>}\ e_0 \mid \underline{\texttt{rec}}\ f\ x\ \texttt{=>}\ e_0 \mid e_0\ \underline{@}\ e_1 \mid \underline{\texttt{if}}\ e_0\ e_1\ e_2 \\ \quad \mid \underline{\texttt{channel()}} \mid \underline{\texttt{fork}}\ e_0 \mid \underline{\texttt{send}}\ e_0\ e_1 \mid \underline{\texttt{receive}}\ e_0 \mid \texttt{lift}\ e \end{array} \tag{3}$$

Underlined expressions are dynamic, or of the second stage, while the other are static, or of the first stage. `lift` c translates a first order static constant into a dynamic one.

The partial evaluation of an expression e is defined by $\mathcal{P}[\![e]\!]\rho$ where ρ is an environment containing global variables (of first and higher order), shared by all the processes. When `Pev` finds a free variable in the program being partially evaluated, it looks for its value in ρ, which can be seen as a memory shared by all the processes and that we use to define the auxiliary functions called by `Pev`.

When `Pev` finds an instruction `fork` e in the program being treated, it creates a new process in which a copy `Pev'` of `Pev` is applied to e. The programs `Pev` and `Pev'` are the same and call the same auxiliary functions which are defined in ρ and that we wish accessible by every process. On the contrary, when a function `fun` x `=>` e is applied, the effective parameter is directly substituted to x in e and ρ is left unchanged. This approach allows us to avoid the problems related to different identically named variables occurring in different processes, as well as problems related to the closure of a communicated function.

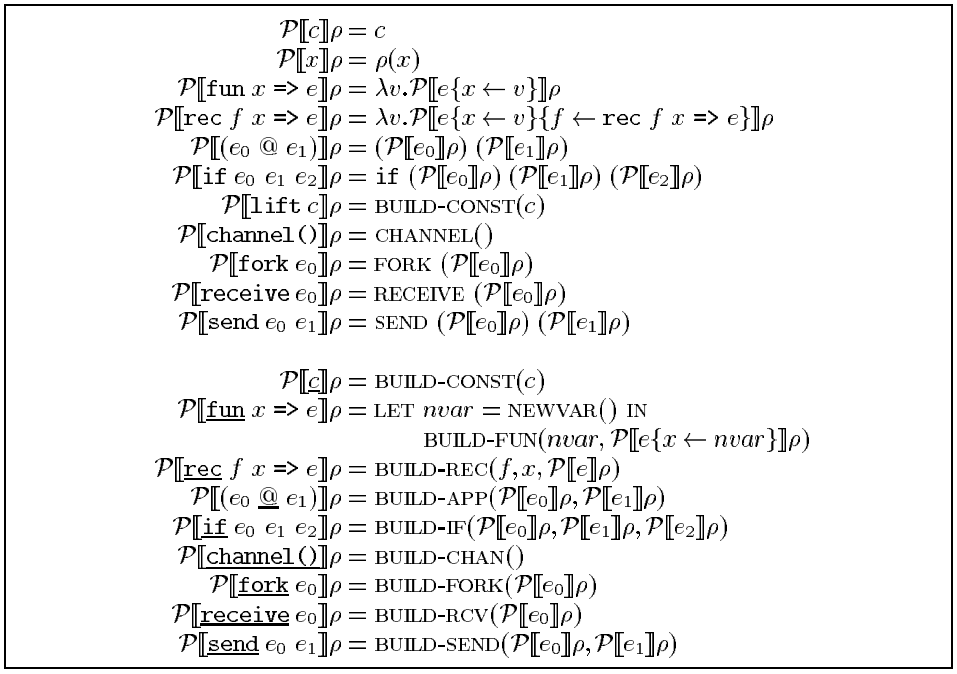

Fig. 1. Evaluation rules of the partial evaluator.

Note that we only define one level of variables. In order to ensure that `Pev` behaves correctly when an unknown variable x is found, we extend the environment ρ by $\rho(x) = \llcorner x \lrcorner$, where $\llcorner x \lrcorner$ denotes the piece of code corresponding to the variable x. Doing so, `Pev` builds a residual piece of code for those variables for which the value is unknown at partial evaluation time.

In Figure 1, we describe the behavior of `Pev`. Functions written in small caps corresponds to operations of the meta-language used to implement `Pev`. Actually, this language is the one of Equation (2) for self-application considerations [11]. The rules used to evaluate sequential expressions are usual [7]. $\mathcal{P}[\![\texttt{channel()}]\!]\rho$ creates a new channel name. The evaluation of $\mathcal{P}[\![\texttt{fork}\ e]\!]\rho$ creates a new process which evaluates $\mathcal{P}[\![e]\!]\rho$ in ρ. When a reception $\mathcal{P}[\![\texttt{receive}\ e]\!]\rho$ is found, e is evaluated in ρ, yielding a result α and, next, the communication `receive` α is done. Similarly, for an emission $\llcorner \texttt{send}\ e_0\ e_1 \lrcorner$, the expressions e_0 and e_1 are evaluated and the results are used to realize the communication.

For second-level expressions, the sub-expressions are partially evaluated and a residual term is built. The functions BUILD-FUN, etc. are auxiliary functions used to build the residual terms. They are defined in the environment ρ. `Pev` executes at partial evaluation time the static communications of a program. These are determined by a binding time analysis introduced in Section ??. Before defining this BTA, we show how `Pev` can be used to partially evaluate some concurrent programs with static and dynamic communications.

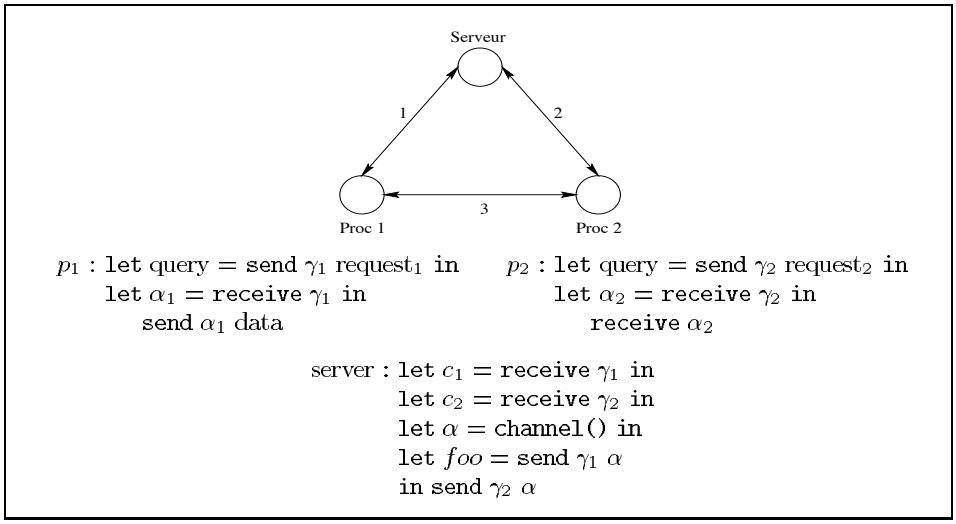

Fig. 2. Creation of a communication link between two processes, via a server.

4 Experiments with Pev

Partial evaluation of concurrent programs allows one to execute at compile-time the static communications of a distributed application. Here we describe some experiments realized with our implementation of **Pev**.

Our first example is given by the program of Figure 2, in which two processes p_1 and p_2 create a communication link between themselves by consulting a server s. p_1 and p_2 are linked to the server by channels γ_1 and γ_2. The channel for the communications between p_1 and p_2 is provided by the server and is named α_1 in p_1 and α_2 in p_2.

We show how to specialize this application for a particular network, i.e. in the case where the channels γ_1 and γ_2 are statically known and in which the server is able at compile-time to determine the communication link that must be used for the communications between p_1 and p_2. However, the data exchanged between the processes p_1 and p_2 are assumed unknown at partial evaluation time.

We model this application by the following program in which the communication channels between the server and the processes are input parameters.

```
define p1-p2 = fun g1 g2 data ->
               let p1 = fork (let query = send g1 p2Id in
                              let a1 = receive g1 in
                                _send a1 data)
               in (* p2 *)   (let query = send g2 p1Id in
                              let a2 = receive g2 in
                                _receive a2) ;
```

The variable `data` is assumed dynamic, so the communication between p_1 and p_2 cannot be achieved at partial evaluation time. This is indicated in the above

program by the symbol _ preceeding the related communication instructions. These annotations can be obtained by the BTA introduced in [11].

We specialize this program in a context where the channels γ_1 and γ_2 are known and where the server is able to compute the communication link to be used for the communications between p_1 and p_2. So, γ_1 and γ_2 are defined and the program is encoded to be understood by the partial evaluator. In addition, we indicate that the variable `data` is unknown. Concurrently with the partial evaluation of the program describing the processes p_1 and p_2, we run the program corresponding to the server, in order to allow the execution of the static communications. This corresponds to the following commands (rewritten for a better understanding).

```
> define ch1 = channel() ;
> define ch2 = channel() ;
> define p1-p2-encoded = encode p1-p2 with g1=ch1, g2=ch2, data=? ;
> run {let server = (fork (let c1 = receive ch1 in
                           let c2 = receive ch2 in
                           let p1p2Ch = channel in
                           let foo1 = send ch1 p1p2Ch
                           in send ch2 p1p2Ch))
       in Pev @ p1-p2-encoded} ;
```

During specialization, the processes p_1 and p_2 are created and the communications with the server are done. So, the effective channel name used as communication link between p_1 and p_2, denoted `#ch`, is inserted in the code of these processes. We obtain the encoding of two residual processes, related to the specialized versions of p_1 and p_2 as shown hereafter.

```
{ send #ch data } | { receive #ch }
```

Our second example is given by the program of Figure 3, which describes a system made of two processes exchanging a message sliced into packets. We assume the packets have a constant size and that their number depends on the size of the message. The process p_1 first sends to the process p_2 the size of the message (assumed, by simplicity equal to the number of packets) and, next, builds and sends the packets. The process p_2 receives the size of the message, realizes as many packet receptions as needed, and rebuilds the message. We specialize this program for the particular case in which the size of the message is known at partial evaluation time, but not the content. The related annotations of the program are given in Figure 3 a) and we assume that the size of the message is 2. Figure 3 b) shows the residual program produced by our partial evaluator. The first communication concerning the size of the message was executed and the loops were unrolled.

Our last example is given by the program of Figure 4 a) which describes a system made of two processes exchanging a message. When the message is received, a checksum is done. If the test fails, which corresponds to a transmission error, a message is sent to the emitter in order to indicate that the message must

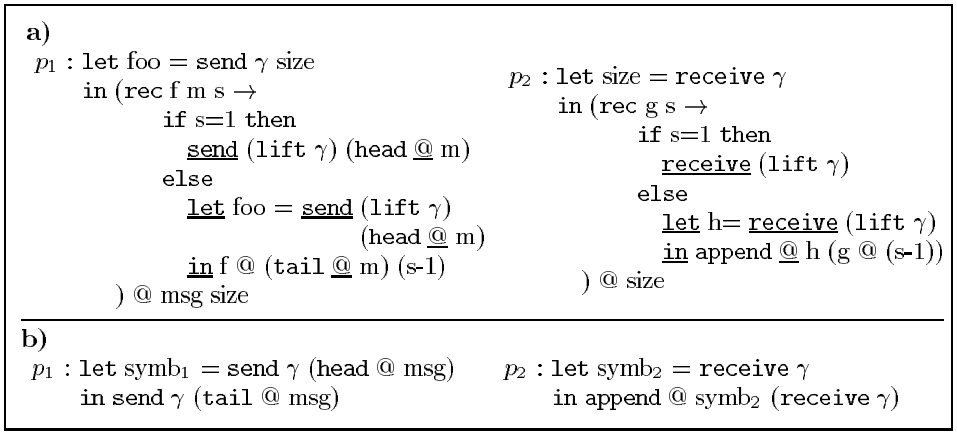

Fig. 3. Slicing of messages into packets of known size and unknown content. a) annotated version of the program. b) Residual program obtained by partial evaluation with size= 2.

be sent again. Otherwise an acknoledgement is sent to the emitter. We specialize this system for a particular network which is assumed error free. We assume that the `checksum` function always indicates that the received message is correct. As shown is Figure 4, b), by partial evaluation, we obtain a new program in which the communications related to the acknoledgements were removed.

5 Conclusion

In this article, we introduced a partial evaluator for programs written in a concurrent functional language and we discussed the results obtained for various examples.

The partial evaluator uses informations about the topology of the communications of the programs. These informations are extracted from the results of a control flow analysis. Even if any CFA can be used, the precisions of this latter and of the BTA are related. The CFA we used is described in [13].

Partial evaluation of concurrent systems allows to specialize applications for particular execution contexts, as shown in the examples of Section 3: static knowledge of the network topology, assumptions on how the networks works or assumptions on the messages. concerning this latter example, we showed how to specialize a communication protocol which slices messages into packets. The specialization was for a fixed size of message. This approach was first introduced by Muller *et al.* who used a partial evaluator to specialize the RPC protocol (Remote Procedure Call) w.r.t. the kind of transmited data [14]. However, due to the fact that the partial evaluator they used only reduces sequential programs, they could not statically execute some communications and had to use run-time specialization techniques instead [4]. Partial evaluation of the communications, as proposed in this article, allows one to statically realize similar specializations.

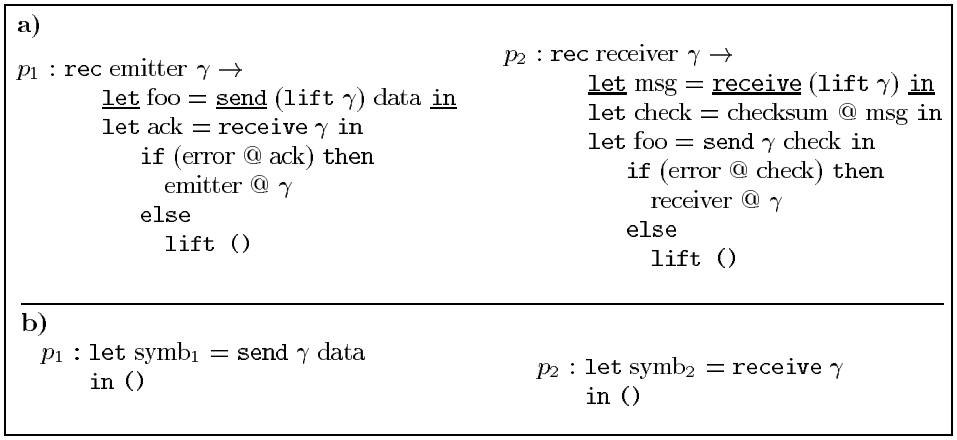

Fig. 4. Partial evaluation of a communication protocol with error detection. a) Annotated version of the program. b) Residual program obtained assuming that the network is error free.

Finally, note that our partial evaluator is compatible with Futamura's projections and automatic compiler generation by self-application of a partial evaluator. The compiler generator related to `Pev` was obtained and is described in [11].

References

1. Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages POPL'92*. ACM, 1992.
2. Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In *Concur'98*, number 1466 in Lecture Notes in Computer Science, pages 84–98. Springer-Verlag, 1998.
3. Anders Bondorf and Jesper Jorgensen. Efficient analyses for realistic off-line parial evaluation: Extended version. DIKU Research Report 93/4, University of Copenhagen, 1993.
4. Charles Consel and François Noël. A general approach for run-time specialisation and its application to c. In *Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages POPL'96*. ACM, 1996.
5. Marc Gengler and Matthieu Martel. Self-applicable partial evaluation for the pi-calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, pages 36–46, 1997.
6. Marc Gengler and Matthieu Martel. Des étages en Concurrent ML. In *Rencontres Francophones du Parallélisme, Renpar10*, 1998.
7. Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.
8. Haruo Hosoya, Naoki Kobayashi, and Akinori Yonezawa. Partial evaluation scheme for concurrent languages and its correctness. In *Euro-par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 625–632. Springer-Verlag, 1996.

9. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993.
10. Mihnea Marinescu and Benjamin Goldberg. Partial evaluation techniques for concurrent programs. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and Semantic based Program Manipulations PEPM'97*, pages 47–62. ACM, 1997.
11. Matthieu Martel. *Analyse Statique et Evaluation Partielle de Systèmes de Processus Mobiles.* PhD thesis, Laboratoire d'Informatique de Marseille, 163 Avenue de Luminy, Case 901F, 13288 Marseille Cedex 9, France, 2000.
12. Matthieu Martel and Marc Gengler. Analyse statique pour la sûreté des applications distribuées. In *Rencontres Francophones du Parallélisme, Renpar12*, 2000.
13. Matthieu Martel and Marc Gengler. Communication topology analysis for concurrent programs. In *SPIN'2000*, volume 1885 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2000.
14. Gilles Muller, Eugen-Nicolae Volanschi, and Renaud Marlet. Scaling up partial evaluation for optimizing the sun commercial rpc protocol. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 101–111. ACM, 1997.
15. Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages.* Cambridge university press, 1992.
16. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.
17. John H. Reppy. An operational semantics of first-class synchronous operations. Technical Report TR-91-1232, Department of Computer Science, Cornell University, Ithaca, NY 14853, 1991.
18. John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.
19. Olin Shivers. *Control Flow Analysis of Higher Order Languages.* PhD thesis, Carnegie Mellon University, Scholl of Computer Science, 1991. Technical Report CMU-CS-91-145.
20. Kirsten L. Solberg, Flemming Nielson, and Hanne Riis Nielson. Systematic realisation of control flow analyses for CML. In *Proceedings of the ACM-SIGPLAN International Conference on Functional Programming, ICFP'97*, pages 38–51. ACM, 1997.